Java Parallel Stream Internals: Demo’ing Spliterator Performance

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change & what can’t
- Partition a data source into “chunks”
  - Know the impact of different Java collections on performance

Starting splitterator tests for 1000 words...
  ..printing results
    17 msecs: ArrayList parallel
    19 msecs: LinkedList parallel

Starting splitterator tests for 10000 words...
  ..printing results
    88 msecs: LinkedList parallel
    90 msecs: ArrayList parallel

Starting splitterator tests for 100000 words...
  ..printing results
    599 msecs: ArrayList parallel
    701 msecs: LinkedList parallel

Starting splitterator tests for 883311 words...
  ..printing results
    5718 msecs: ArrayList parallel
    31226 msecs: LinkedList parallel

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14
Demonstrating Spliterator Performance
Demonstrating Spliterator Performance

- Spliterators for ArrayList & LinkedList partition data quite differently

See earlier lesson on “Java Parallel Streams Internals: Partitioning’
Demonstrating Spliterator Performance

- Spliterators for ArrayList & LinkedList partition data quite differently

```java
ArrayListSpliterator<E> trySplit() {
    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    // divide range in half unless too small
    return lo >= mid ? null : new ArrayListSpliterator<E>(list, lo, index = mid, ...);
}
```

ArrayList’s spliterator splits evenly & efficiently (e.g., doesn’t copy data)

See [openjdk/8u40-b25/java/util/ArrayList.java](openjdk/8u40-b25/java/util/ArrayList.java)
Demonstrating Spliterator Performance

- Spliterators for ArrayList & LinkedList partition data quite differently

```java
Spliterator<E> trySplit() { 
    int n = batch + BATCH_UNIT, j = 0; Object[] a = new Object[n];
    do { a[j++] = p.item; }
    while ((p = p.next) != null && j < n); 
    return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
}
```

- LinkedList’s spliterator does not split evenly & efficiently (e.g., it copies data)

See openjdk/8u40-b25/java/util/LinkedList.java
Demonstrating Spliterator Performance

- This demo program shows the performance difference of parallel spliterator for ArrayList & LinkedList when processing the complete works of Shakespeare.

```java
void timeParallelStreamUppercase(String testName,
                                   List<CharSequence> words) {

    List<String> list = new ArrayList<>();

    for (int i = 0; i < sMAX_ITERATIONS; i++)
        list.addAll(words.parallelStream()
                  .map(charSeq -> charSeq.toString().toUpperCase())
                  .collect(toList()));

    ...
```
Demonstrating Spliterator Performance

- This demo program shows the performance difference of parallel spliterators for ArrayList & LinkedList when processing the complete works of Shakespeare.

```java
void timeParallelStreamUppercase(String testName,
        List<CharSequence> words) {
    ...  
    List<String> list = new ArrayList<>();
    for (int i = 0; i < sMAX_ITERATIONS; i++)
        list
            .addAll(words
                .parallelStream()
                .map(charSeq ->
                    charSeq.toString().toUpperCase())
                .collect(toList()));  ...
```

The words param is passed an ArrayList & a LinkedList.
This demo program shows the performance difference of parallel spliterators for ArrayList & LinkedList when processing the complete works of Shakespeare.

```java
void timeParallelStreamUppercase(String testName,
                                 List<CharSequence> words) {
    ...
    List<String> list = new ArrayList<>();

    for (int i = 0; i < sMAX_ITERATIONS; i++)
        list.addAll(
            words
                .parallelStream()
                .map(charSeq ->
                    charSeq.toString().toUpperCase())
                .collect(toList())); ...
```

**Split & uppercase the word list via a parallel spliterator**
Results show spliterator differences become more significant as input grows

Starting spliterator tests for 1000 words....printing results
  17 msecs: ArrayList parallel
  19 msecs: LinkedList parallel

Starting spliterator tests for 10000 words....printing results
  88 msecs: ArrayList parallel
  90 msecs: LinkedList parallel

Starting spliterator tests for 100000 words....printing results
  599 msecs: ArrayList parallel
  701 msecs: LinkedList parallel

Starting spliterator tests for 883311 words....printing results
  5718 msecs: ArrayList parallel
  31226 msecs: LinkedList parallel

See upcoming lessons on "When [Not] to Use Parallel Streams"
Demonstrating Spliterator Performance

```java
private static void runSplitteratorTests() {
    // Create tests for different sizes of input data.
    .asList(1000, 10000, 100000, 1000000)

    // For each input data size run the following tests.
    .forEach(limit -> {
        // Create a list of strings containing all the
        // words in the complete works of Shakespeare.
        List<CharSequence> arrayWords =
            TestDataFactory.getInput(sSHAKESPEARE_DATA_FILE,
                // Split input into "words" by
                // ignoring whitespace.
                splitter: "\s+",
                limit);

        // Create a LinkedList from the ArrayList.
        List<CharSequence> linkedWords =
            new LinkedList<>((arrayWords);
    })
}
```
End of Java Parallel Stream Internals: Demo’ing Spliterator Performance